Optimizing Shuffle Performance in Spark

Aaron Davidson
UC Berkeley

Andrew Or
UC Berkeley

ABSTRACT

Spark [6] is a cluster framework that performs in-memory computing, with the goal of outperforming disk-based engines like Hadoop [2]. As with other distributed data processing platforms, it is common to collect data in a many-to-many fashion, a stage traditionally known as the shuffle phase. In Spark, many sources of inefficiency exist in the shuffle phase that, once addressed, potentially promise vast performance improvements. In this paper, we identify the bottlenecks in the execution of the current design, and propose alternatives that solve the observed problems. We evaluate our results in terms of application level throughput.

1. INTRODUCTION

In traditional MapReduce frameworks, the shuffle phase is often overshadowed by the Map and Reduce phases. In fact, shuffling is commonly integrated as part of the Reduce phase, even though it really has little to do with the semantics of the data. However, shuffling data in a many-to-many fashion across the network is non-trivial. The entire working set, which is usually a large fraction of the input data, must be transferred across the network. This places significant burden on the OS on both the source and the destination by requiring many file and network I/Os. To achieve high performance, distributed coordination is important for load balancing purposes. Especially under big data workloads, this is a known problem. [1]

1.1 Existing Solutions

Several solutions have been proposed to alleviate the stress placed on the OS. Because the shuffle phase is agnostic to the semantics of the data, an extra processing stage can significantly reduce the size of the data actually transferred. Compression of Map output files before they are shuffled across the network is popular in most MapReduce frameworks. Another common technique is to use combiners, which begin reducing on the Map side as soon as the Map output is ready.

However, the effectiveness of both techniques is highly dependent on the structure of the input data and the application. In particular, compression is much less effective on arbitarily formatted text than on primitive types in key-value pairs. Many applications perform simple arithmetic reductions on the Map output, in which case combiners serve to lessen the load of shuffling by reducing the amount of data transferred. In contrast, in applications that concatentate or even amplify Map outputs, combiners may end up inflating the amount of data shuffled, rather than deflating it.

Both of these solutions act on a very high level, and are general enough to be applicable to all MapReduce frameworks. However, these frameworks are ultimately very different systems that provide different use cases and offer different properties. A small implementation detail in one system can lead to vast performance differences when compared to another system. Prior work [1] has demonstrated that Hadoop faces completely different performance characteristics under different sets of parameters on both the Map side and the Reduce side. Thus, in optimizing the performance of such frameworks, it is crucial to identify the bottlenecks and instrument an implementation specific to each system.

1.2 Spark

We choose to optimize shuffle file performance in the Spark distributed computing platform. The underlying reason for our choice is threefold: first, Spark is not only open-source, but also relatively young. This allows us to propose changes much more easily than a more mature system like Hadoop, the framework that popularized the MapReduce model in the first place. Additionally, because Spark's shuffle performance has not received nearly as much attention as other systems' have, there is more opportunity in instrumenting a significant improvement.

Second, an early evaluation of Spark against Hadoop in a typical workload that does not take advantage of Spark's in-memory computing reveals that Spark's performance is actually subpar. Part of the motivation of the paper is to understand the reason behind this, and implement an optimization that at worst narrows the discrepancy in performance between the two systems under these workloads.

Third, Spark, originally a research project that resided in the academic space, is gaining attention in the industry

as the new generalized system which may one day replace Hadoop MapReduce, the current industry standard. This creates demand for Spark to have performance characteristics no worse than the existing status quo.

The rest of the paper is organized as follows. Section 2 describes prior work relevant to our project. Section 3 details how other systems differ from Spark in terms of shuffling, and how the bottlenecks observed are specific to Spark. Section 4 describes our approach in mitigating these bottlenecks, and Section 5 evaluates the changes we propose. Finally, in Section 6, we discuss how future work complements what is currently lacking in our instrumentation.

2. RELATED WORK

The problems inherent in the Spark shuffle phase are shared among many other distributed systems. At its heart, the shuffle phase can variably stress the CPU, memory, disk, and network capacities of the cluster; any one of these aspects may become a bottleneck for the computation.

One of the most complete attempts at a solution to this problem is TritonSort [3], whose explicit goal is to eliminate the bottleneck issue by attempting to ensure that all computational aspects are bottlenecked simultaneously. Unfortunately, due to the authors' highly specific problem choice of sorting well-distributed data and the well-behaved, carefully laid-out compute cluster, their optimizations do not easily generalize to Spark, which must explicitly work on heterogeneous datasets and node distributions. For example, they assume that every worker is processing data at an equal rate, but stragglers in MapReduce are not only possible, but very common.

While the solutions presented in TritonSort are difficult to generalize, the problems discovered during the analysis of their system are still highly relevant. Per-node network bandwidth is limited, and the high-bandwidth all-to-all communication required by a shuffle operation is further inhibited by TCP incast [1], network queue buildup, and limited memory for buffers. Additionally, disk throughput decreased noticeably (25%) after writing one stage (of a 2-phase sort) to disk. Though the authors attribute this phenomenon to the lack of remaining space of the inner sectors of the disks, we saw a very similar disk performance problem in Spark due to file buildup, discussed in Section 3.

Another paper which took a careful look at the cost of large scale data movement is the C-Store column-oriented distributed DBMS [5]. In particular, C-Store examined the potential gain of compressing individual columns of structured data, as opposed to the row-based compression possible in traditional DBMSs. As compression is essentially the process of finding patterns within data, it is clear that columns individually are more amenable to compression than the alternate representation of a row, due to the increased type and content homogeneity within a single column. C-Store was additionally able to achieve further compressibility by sorting columns individually. While the Spark data model is more general than C-Store's structured data model – enabling arbitrary Java objects to act as elements of an RDD – it is very common to use tuples (e.g., key-value pairs), a property that allows us to examine columnar compression as an optimization to the Spark shuffle phase (Section 4).

3. BACKGROUND

In this section, we discuss the approaches taken by existing MapReduce systems in optimizing the performance of the shuffle phase, and extend the relevance of these approaches to Spark. We observe that the bottleneck that Spark currently faces is a problem specific to the existing implementation of how shuffle files are defined.

3.1 Hadoop

3.1.1 Map

When a Map task finishes, its output is first written to a buffer in memory rather than directly to disk. Only after the buffer exceeds some threshold does it spill to disk. The outputting of Map results to disk is therefore specified by two parameters: `io.sort.mb`, the size of the in-memory buffer, which defaults to 100MB, and `io.sort.spill.percent`, the threshold of the buffer before its content is spilled to disk, which defaults to 80%. Thus, by default, if the input of a Map task is 10GB, then we end up with $\frac{10GB}{100MB * 80\%} = 125$ intermediary shuffle files per Map task.

However, it is common for a fraction of the input to be shaved before being written to a shuffle file. As an example, a tweet contains many fields, many of which may not be relevant to a particular MapReduce job, in which case the irrelevant fields are discarded, leaving only the ones of interest.

Further, the remaining data can also be compressed, enabled by setting `mapred.compress.map.output` to `true`. The compression library, specified by `mapred.map.output.compression.codec`, can be one of gzip, bzip2, or LZO, each of which have different compression ratios and speeds. Additionally, a user-defined combiner function specifies how the Map output can be combined, as long as the number of spills is at least the value of `min.num.spills.for.combine`, which defaults to 3.

Finally, for each Map task, rather than sending the output files as spilled files, Hadoop merges them into a single file in sorted order, partitioned by the number of reducers on the receiving end. The maximum number of streams to merge at once is governed by `io.sort.factor`, which

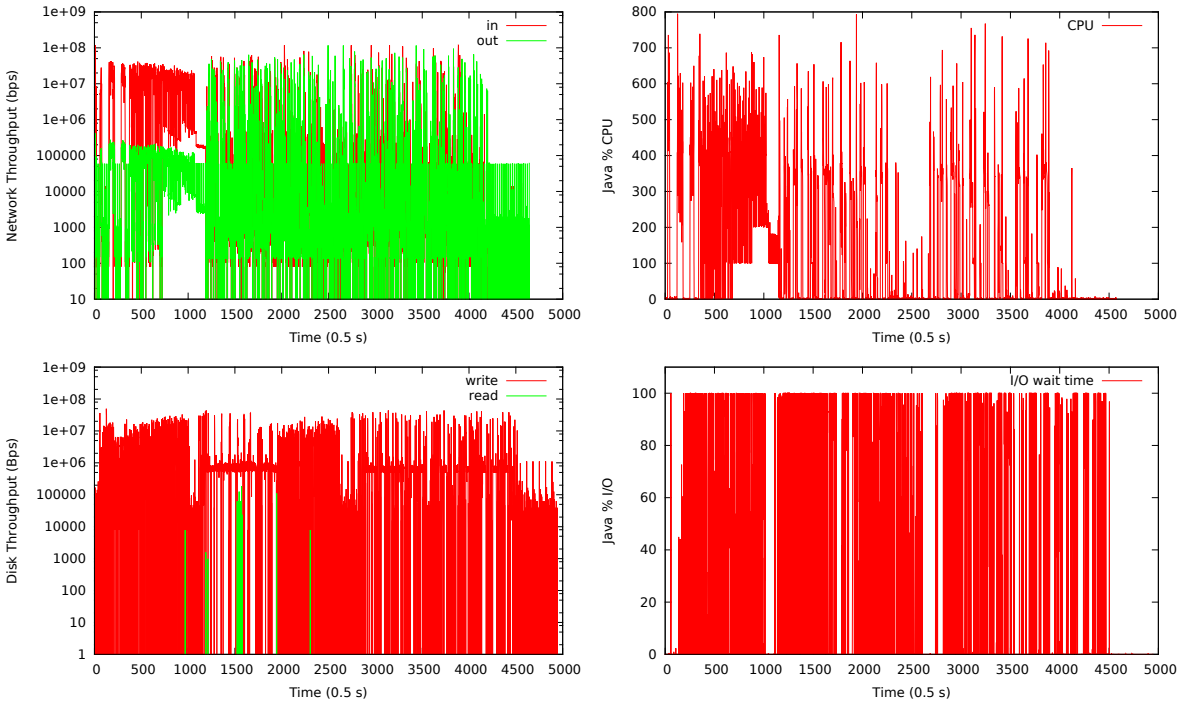

Figure 1: Network, CPU, and I/O characteristics in Spark (before)

defaults to 10.

3.1.2 Reduce

Although the Reduce phase is distinct from the Map phase in terms of functionality, these two stages overlap in time. During the *copy phase* of the Reduce task, each Map task informs the *tasktracker* as soon as it finishes, and then *pushes* its output to the appropriate Reduce task. The Reduce task can copy from up to `mapred.reduce.parallel.copies` threads at once, 5 by default. Importantly, *tasktrackers* do not delete Map output as soon as the transfer is complete, but instead keep them persisted in disk in case the reducer fails.

As in the Map phase, the Reduce phase also maintains an in-memory buffer for shuffle files, governed by `mapred.job.shuffle.input.buffer.percent`, the percentage of heap space for this buffer, defaulting at 70%. The content of the buffer is spilled to disk when at least one of two things happens: the percent of buffer occupied exceeds `mapred.job.shuffle.merge.percent` (66% by default), or when the number of Map outputs exceeds `mapred.inmem.merge.threshold` (1000 by default).

This spilled files are also merged into fewer, larger files, as in the Map phase. The maximum merge factor is also specified by `io.sort.factor`. The final merge needs not be from disk alone; it can read from a mixture of in-memory and on-disk segments. The sorted output is now ready to be streamed into the reduce function.

3.2 Spark

3.2.1 Map

Instead of maintaining a common in-memory buffer, Spark Map tasks write their output directly to disk on completion, relying on the operating system's disk buffer cache to avoid an excess amount of disk writes. More specifically, each Map task writes one shuffle file per Reduce task, which corresponds to the logical Block in Spark.

The concept of a Block here is reused from the more general Spark data model, where all data sets (RDDs and shuffle output) are composed at the finest granularity by individual Blocks. Shuffle files originally were decomposed into Blocks in order to share the same management, directory, and transferral code with RDDs – however, in Spark today there is a separate code path for all of these operations, for performance reasons.

Thus, each map task writes R shuffle files, where R is the number of Reduce tasks. Unlike Hadoop, however, these are not intermediary files, as Spark does not merge them into a single, partitioned one. It is worth noting that, in general, both M and R are often larger in Spark than in Hadoop due to Spark's lower scheduling overhead per task. In practice, M could be 5000 and R could be 1024,

amounting to over 5 million shuffle files in total. As we shall see, the sheer number of shuffle files written is a major source of performance degradation.

As in Hadoop, Spark provides the option to compress Map output files, specified by the parameter `spark.shuffle.compress`. The compression library, specified by `spark.io.compression.codec`, can be by default Snappy or LZF. At the same time, however, compression is also potentially a source of memory concerns. In particular, LZF requires 400KB of buffer for each open file, multiplied by R per Map task. Since, at any given time, there are C concurrent Map tasks per machine, where C is the number of cores allocated to Spark, the amount of memory used by LZF is 400KB * RC. For example, with $R = 1024$ Reduce tasks and $C = 8$ cores, this becomes 3.2GB per machine. Under memory constraints, Snappy, which uses only 33KB of buffer for each open file, can greatly reduce the risk of running out of memory.

3.2.2 Reduce

A stark difference between Spark and Hadoop in the Reduce phase is that Spark requires all shuffled data per Reduce task to fit into memory when the Reduce task demands it. This could happen if the Reduce task involves a groupByKey, or a reduceByKey that, for instance, concatenates values. When the memory required of each Reduce task exceeds what it is allocated, then an out of memory exception is thrown and the entire job must be aborted. To avoid running out of memory on the Reduce side, the application must specify a high enough value for R, possibly through trial and error. The lack of a spilling mechanism in Spark is in fact a major problem faced by many users.

Another important difference between the two systems is that the Spark Reduce phase does not overlap with the Map phase. In other words, shuffling in Spark is a *pull* operation, rather than a *push* operation as in Hadoop. This implies that the collective memory required by all concurrent Reduce tasks must be available at any given time of the Reduce phase, since the Reduce phase is no longer spaced out over time.

In addition, each Reduce task must maintain a network buffer for fetching Map outputs. The size of this buffer is specified by `spark.reducer.maxMbInFlight`, with a default value of 48MB. The strain on memory contributed by this buffer is negligible, however.

3.3 Identifying Bottlenecks in Spark

To investigate the bottlenecks faced by Spark under typical workloads, we ran a simple job that finds the most popular text in a tweet, given an input dataset of 1TB distributed across 16 m2.4xlarge EC2 nodes. (More details in Section 5.)

The results are shown in Figure 1. The network is clearly not the bottleneck; at any given point, the throughput of both outgoing and incoming traffic is on the order of 10Mbps, even though the available bandwidth is on the order of 1Gbps. Neither is CPU; each node has 8 cores, such that if the job was CPU-bound, then the %CPU used by Spark would consistently hover slightly below 800%. This is not the case in our experiment. Similarly, disk throughput is on the lower order of 10MBps, which is still nowhere near the upper bound at around 80MBps. On the other hand, the % time of Spark waiting on disk I/O is not only very high (near 100%), but also subject to high variability. This suggests that Spark performance was mainly suffering from heavy random I/Os.

It is worth noting that these results verify the hypothesis that a large number of shuffle files imposes a heavy load on the operating system. In the previous subsection, we demonstrated that the total number of shuffle files can be on the order of millions. Although this can be mitigated by increasing the number of nodes such that the stress from creating many files can be diffused across the nodes, each machine is still responsible for a sizeable number of shuffle files.

During the Map phase, we hash shuffle files into a set of 64 subdirectories created on each disk. This can cause a large number of random writes when each shuffle file is relatively small. In the Reduce phase, there are two possible sources of random reads. First, requests for individual shuffle blocks arrive in a random order as they are requested concurrently by all executing reducers. Second, if the inode cache is too small to hold the working set of all shuffle files, then an extra random read for the inode will be incurred for each read of a shuffle block.

One solution is to emulate Hadoop's behavior by writing out a set of smaller files to the same directory and subsequently merging them in a second pass. Another is to create larger shuffle files in the first place. Both solutions would reduce the number of random writes during the Map phase as well as the number of inodes necessary to index the shuffle blocks. The first solution requires both more modification to Spark as well as a second pass over all shuffle data. Therefore we mainly investigated the second solution.

4. OUR APPROACH

4.1 Columnar Compression

Following the success of C-Store's columnar compression, we attempted to apply the same scheme to Spark's shuffle phase in order to offload some of the burden from the network and disk onto the CPU. As Spark's data model is very general, allowing data to be arbitrary serializable Java objects, it is not possible to decompose data into columns in general. However, from common usage patterns, we ob-

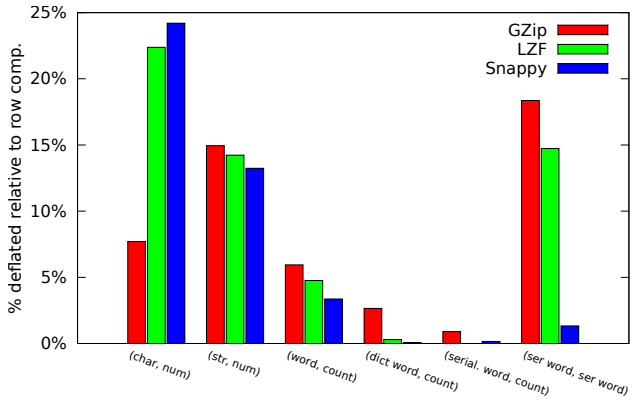

Figure 2: Columnar Compression

serve that it is very common to use a tuple of primitive data types or Java objects during the shuffle phase. This is especially typical in jobs involving shuffles, as Spark requires data to be a tuple of (key, value) pairs in order to perform aggregation or joins.

In order to assess the potential benefit of columnar compression, we ran a microbenchmark that compares the compressibility of a set of realistic key-value examples taken from the Spark website and user group, when the keys and values were compressed both individually (column-based) and together (row-based). Figure 2 shows the gain of column compression relative to row compression for our examples using multiple compression schemes. Surprisingly, column compression was rarely significantly better than row compression, and occasionally worse. We attribute this to two factors specific to Spark.

First, the most compressible portion of the data is usually not the biggest contributor to overall size in the first place. For example, in the common word count example, tuples are of the form (word, 1), so they can be easily aggregated by summation over the values. Using run-length encoding, we can use an extremely small number of bytes for the count column – however, the majority of the data is present in the much less compressible "word" column, so the gain is not particularly impressive.

Second, serialized objects tend to have a lot of repeated overhead per object. However, the use of dictionary compression is equally effective at reducing this overhead when the objects are in individual columns as when they are combined, since the dictionary replacement needs to be performed once per object either way. All of the examined compression algorithms utilize a dictionary.

Due to the lack of significant results for using columnar compression, we did not investigate it further. In addition to the unimpressive compressibility, splitting data into columns on the map side and reconstructing them back into rows on the reduce side requires a significant amount of additional metadata and computation. It is possible that column-oriented reduction algorithms like those presented in C-Store (i.e., methods that don't require reconstructing rows) could be used to mitigate the computational overhead, but we did not investigate these.

4.2 Shuffle File Consolidation

In our analysis in Section 3, we demonstrated that Spark creates a large number of shuffle files ($M*R$) and verified that this number placed a significant strain on the operating system, on both the Map phase and the Reduce phase. In this subsection, we pursue one of the alternatives we proposed: rather than instrumenting Spark to spill contents of an in-memory buffer, thus introducing an additional merge phase, our solution is to write fewer, larger files in the first place.

The main reason behind this decision is a practical one. Introducing an extra phase involves a much more significant overhaul of the Spark workflow. Even after introducing a radical change in the system, there is no guarantee that Hadoop's workflow is by nature superior to Spark's. In fact, the values of M and R in Hadoop are usually much smaller, as we described previously. Since the number of shuffle files in Spark currently scales with $M*R$, a smaller number of Map tasks and Reduce tasks may provide more justification for the way Spark handles shuffle files on the Map side.

At the same time, it is important to observe that we cannot simply lower these values in the Spark configuration due to memory constraints previously discussed. Thus, it is our decision to not simply adopt the strategies of other mature MapReduce frameworks, but rather investigate a solution more specific to Spark. Our goal is to optimize Spark shuffle performance, rather than to investigate the performance characteristics of a radically redesigned workflow, a project that is no less interesting but far beyond the scope of this paper.

Shuffle file consolidation refers to maintaining a shuffle file for each partition, which is the same as the number of Reduce tasks R, per core C rather than per Map task M. In other words, all Map tasks running on the same core write to the same set of files in tandem, one for each Reduce task. The scaling property of consolidating shuffle files this way is much more desirable. Every machine needs to handle only $C*R$ number of shuffle files rather than $M*R$. More concretely, given realistic numbers of $C = 8$, $M = 5000$, and $R = 1024$, shuffle file consolidation scales the number of files from 5000 * 1024 = over 5 million to 8 * 1024 = 8196, a decrease of around 3 orders of magnitude.

To observe the differences in performance, we ran a pre-

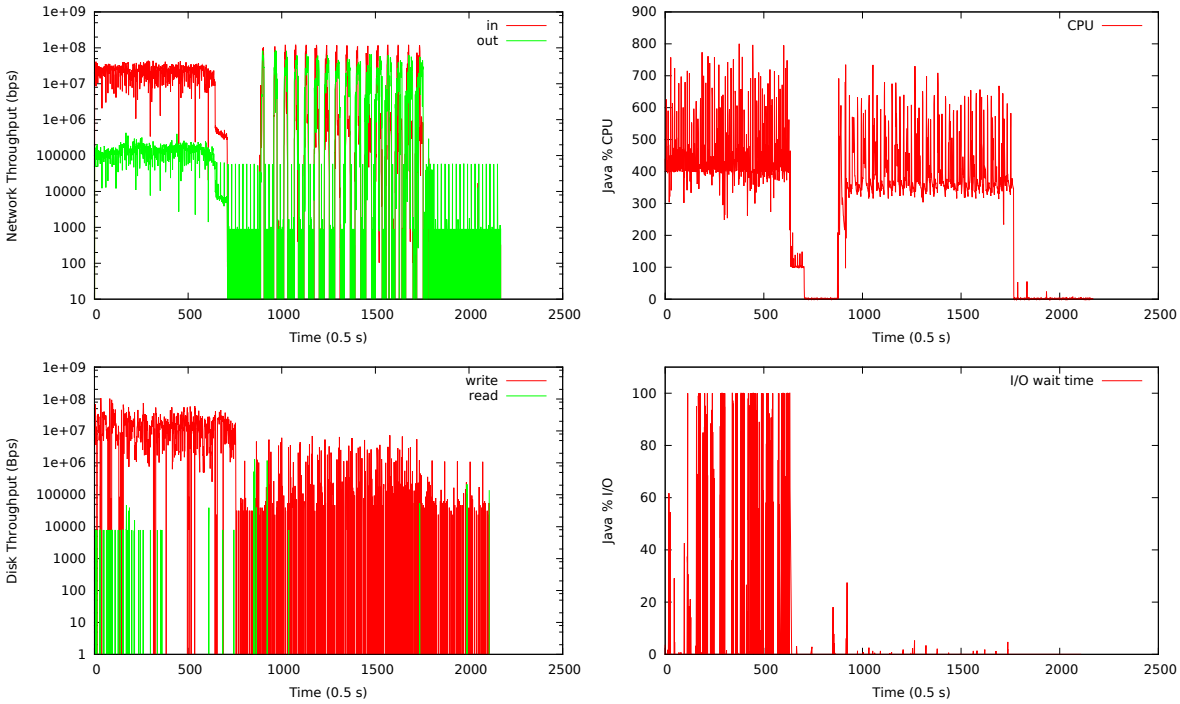

Figure 3: Network, CPU, and I/O characteristics in Spark (with shuffle file consolidation)

liminary experiment that is identical to the one we ran for Spark before applying shuffle file conslidation. The results are shown in Figure 3. The most significant improvements are observed during the Reduce phase. In particular, the I/O wait time is much lower than before, and the disk throughput is sustained on the order of 1MBps. Further, the CPU is also much more static above 400%, implying that less idle time is spent waiting for I/O. All of these observations collectively suggest that Spark does much less random I/O by creating fewer shuffle files.

More concretely, during the Reduce phase, each node fetches a fewer number of shuffle files from each other node. Because of the way the shuffle files are hashed into multiple directories, they tend not to be co-located on disk. Further, even if the shuffle files are written sequentially to disk during the Map phase, each Reduce task fetches only the partitions that correspond to itself. This order of fetching is very different from the order of writing the shuffle files in the first place, which implies that random I/O's during shuffle file fetches are inevitable given the order of execution of Map tasks and Reduce tasks in Spark.

Nevertheless, this provides a stronger case for reducing the number of shuffle files to be fetched. This is especially important in Spark because the Reduce phase begins only after the Map phase has finished, suggesting that remote fetches on one machine may be synchronized among all executing Reduce tasks on other machines, inflating the aggregate random access throughput required.

This alone, however, does not explain the vast differences in the I/O wait time observed in Figure 3. We suspect that this is because the in-memory inode cache is now able to provide much stronger locality. With many shuffle files, the inode cache cannot possibly maintain all metadata in memory. As a result, the working set of inodes maintained in the inode cache is only a small fraction of what is required to read all shuffle files from disk. This leads to an additional random I/O for the inode in each shuffle file fetch, amplifying the stress on the operating system during the shuffle phase.

With much fewer shuffle files, we conjecture that most, if not all, the inodes in the working set of shuffle files can now fit into the file system's inode cache. Thus, maintaining a smaller number of shuffle files not only incurs fewer shuffle file reads, but also allows Spark to exploit stronger locality benefits offered by the underlying file system.

Beyond the conclusions we draw from these graphs, a few anomalies can be identified, and here we make an attempt to explain them. In terms of CPU, there is a lull between the Map phase and the Reduce phase. This could simply be due to the variability in the nature of any experiment run in a distributed environment. More specifically, the

Virtual cores	8
RAM	68.4GB
Disks	2x 840GB
Network	1 Gbps

Table 1: Experiment Configurations

particular node from which we draw these data from is probably held back by other stragglers. Immediately before the lull, the CPU usage actually briefly stepped down from above 400% to 100%. We attribute this behavior to the fact that Spark speculation transfers Map tasks from slow nodes to fast nodes that have already completed their original allocation. Then, the momentary step down in CPU usage may be the result of this node suddenly taking over a small number of tasks from other stragglers after completing its own at full speed.

Similarly, there is a lull in network throughput between the Map phase and the Reduce phase. The explanation for this is the same as the one given for the lull in CPU usage. A more interesting observation in the network graph is the behavior during the Reduce phase. In particular, there are exactly 16 spikes in the network throughput of both the outgoing and incoming connections. Each spike most likely represents a fetch from each node, as there are exactly 16 nodes in the cluster in this particular experiment. The variability of network throughput is also much lower in both phases. While we do not have a satisfactory explanation for this, we suspect it is related to the fact that each node is now spending less time performing random I/Os during shuffle file read and writes, thereby allowing network traffic in either direction to be less volatile.

5. EVALUATION

5.1 Experimental Setup

We ran our tests on Amazon Cloud Compute clusters. We used 16 m2.4xlarge instances (details shown in Table 1). All nodes were located in the us-east-1 region (Virginia) and the same Availability Zone. Maximum observed disk throughput was around 80 MB/s and maximum inter-node network throughput was 120 MB/s.

In all workloads, our initial data was stored in Amazon S3. During a microbenchmark, each node was able to fully saturate its inbound network connection by reading from S3. Intermediate shuffle data was split between the two physically attached disks.

5.2 Workloads

We examined two workloads (Table 2) at opposite ends of the shuffle file spectrum. Both workloads were based on real data and use-cases from a Spark user. In both cases, the data was composed of a set of JSON records. The job

	Workload 1	Workload 2
Input data size	100GB	1TB
Total shuffle data size	102GB	0.9TB
Compressed shuffle data	78GB	80GB
# Mappers	5052	8192
# Reducers	2048	512

Table 2: Experiment Workloads

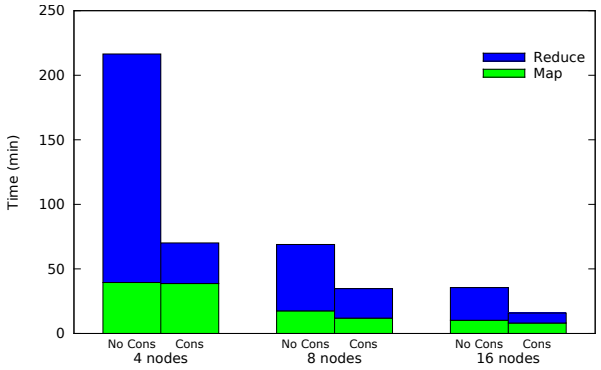

Figure 4: Spark shuffle file consolidation scaling with Workload 1

parsed the record, split it based on one field, performed a group-by-key operation (invoking a full data shuffle), and then did some minor post-processing. One workload consisted of data that was highly compressible, allowing CPU to be used to greatly reduce disk and network bandwidth. The other workload was not compressible, meaning compression had very little impact and almost all of the input data had to be shuffled across the network.

We ran our experiments over both workloads, but have omitted the results for Workload 2 in many cases as they mirrored the results for Workload 1. We felt that workload 1 was more generalizable, considering the extraordinary compressibility of shuffle files in workload 2 (1 TB to 80 GB).

5.3 Experiment 1: Scaling

Hadoop is well-known for having linear strong scaling, which is made possible by its simple shared-nothing model. Spark seeks to emulate the scaling of Hadoop, despite its increased complexity and out-of-band coordination. In order to examine the impact of the OS file system performance degradation on the scaling of the number of nodes, we ran the same job with the same amount of data on 4, 8, and 16 nodes. Each run used LZF compression, which is the Spark default, with 58 GB of RAM allocated in total between the 8 cores. The results from this experiment are shown in Figures 4 and 5, for Workloads 1 and 2 respectively.

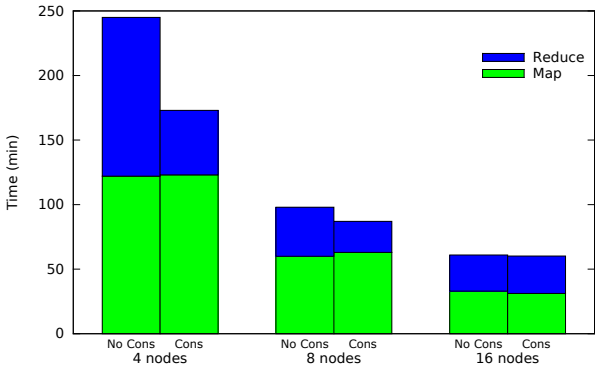

Figure 5: Spark shuffle file consolidation scaling with Workload 2

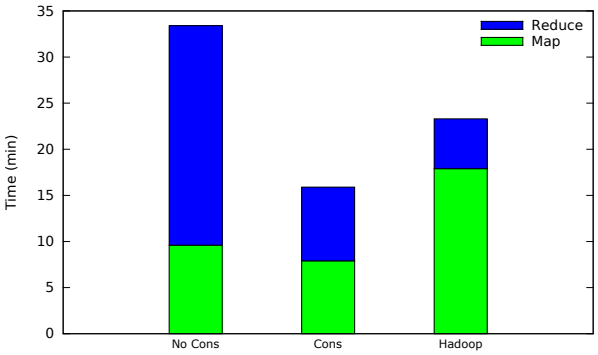

Figure 6: Performance of Hadoop and Spark, with and without shuffle file consolidation

One facet of the file system degradation issue that we immediately discovered was that the problem does not appear at all until the number of shuffle files reaches around one million, and subsequently becomes much worse as the number of files increases further. For example, it takes around 35 minutes to run Workload 1 on 16 nodes without consolidation, but it takes over 6x longer on 4 nodes, rather than the expected 4x degradation. Figure 4 shows that shuffle file consolidation fixed this scaling problem, such that Spark now demonstrates linear strong scaling. Figure 4 additionally shows that consolidation led to around a 2x improvement on Workload 1, even for 16 nodes.

This 2x improvement is possible because even with 16 nodes, there were over 600,000 shuffle files per machine, which did trigger the file system slowdown. Compare these results to the results of Workload 2 in Figure 5, which had significantly fewer shuffle files. As the slowdown was not triggered in the 16 node case, the runtimes of with-consolidation and without-consolidation were approximately equal. Only when the number of files became sufficiently high in the 8 and 4 node runs did performance degrade significantly.

Additionally, it is of note that the Map phase was roughly the same between runs with and without consolidation and always experienced strong linear scaling; only the Reduce phase experienced a slowdown and poor scaling. Examination of the stacks of executing Reduce tasks during without-consolidation runs suggests that a large portion of time was spent waiting for file system metadata-related operations, such as File.length(). This, combined with the high I/O wait times experienced during the Reduce phase (which were eliminated with consolidation) suggest that the inode data must be retrieved from disk during the Reduce phase, which is likely causing a large part of the slowdown.

5.4 Experiment 2: Hadoop

The particular job we examined – consisting of reading data from S3, parsing it, performing a cogroup, and writing it to HDFS – is in reality a workload more suitable for Hadoop than Spark. As a one-time bulk job, we could not take advantage of Spark's in-memory caching, its low-overhead recovery mechanism, its reduced scheduling overhead, or its higher level primitives. Nevertheless, we were interested in how Spark compares to Hadoop.

This experiment was run on all 16 nodes using Workload 1 with no compression enabled (as Spark and Hadoop don't share default compression schemes, and this is not the highly compressible workload anyway). We compared our normal Spark job with a completely analogous one written in Scalding [4], which is a Scala library that compiles into Hadoop jobs, similar to Pig. As both jobs were written in Scala and with APIs that look very similar, we are confident in the equivalence of the two programs in terms of high level operations.

To our surprise, the Spark job with consolidation actually outperformed Hadoop, while Spark without consolidation performed significantly worse (Figure 6). There are two possibilities on why Hadoop may have performed worse here. First, simple misconfiguration; through the course of our work we became very familiar with the Spark settings that had to be tuned; on the other hand, we only ran Hadoop a relatively small number of times before it worked with reasonable performance. This does speak for Hadoop's relative ease of configuration: the initial configuration did work, just with poor performance. On the other hand, Spark's initial configuration would throw out-of-memory exceptions at the beginning of the Reduce phase due to having too few reducers, so tuning was not only important, but also necessary.

A second factor in Hadoop's performance is, naturally, its more expensive shuffle operation compared to Spark. As this job is almost entirely just a shuffle, Hadoop's ex-

	No consolidation		Consolidation	
File system	ext3	ext4	ext3	ext4
Map phase	9.5 min	10.2 min	13.9 min	8.1 min
Reduce phase	7.8 min	25.3 min	19.7 min	7.7 min
Total	17.3 min	35.5 min	33.6 min	15.8 min

Table 3: Performance of Spark on ext3 vs ext4 (16 nodes)

ternal sort-merge approach costs significantly more CPU and disk bandwidth. We believe this was the main cause of Hadoop's Map phase being significantly slower than Spark's, both with and without consolidation.

On the other hand, Hadoop does have an optimization that Spark currently omits: shuffle data is prefetched by reducers while the Map phase is still running. This accounts for the smaller time spent in the Reduce phase in Hadoop's case. In reality, the Reduce phase overlaps with the entirety of the Map phase; the segmented part in Figure 6 shows only the time spent after the map phase ended. We would have initially predicted this prefetching to have a very good performance impact for this job, which is bound by its shuffling. However, we found that the Hadoop task was actually bound by CPU, similar to Spark, which was exacerbated by the need to sort all of the shuffle data.

5.5 Experiment 3: File Systems

Shuffle file consolidation aims to ease the random I/O access overheads. Therefore, the costs associated with these access patterns are highly dependent on the underlying file system. So far, all of the results presented are derived from running Spark on nodes with ext4. In this experiment we run the same jobs on both ext3 and ext4 in hopes of identifying interesting patterns.

Table 3 presents the duration of each phase in both ext3 and ext4, with and without shuffle file consolidation. The configurations used in this experiment are as follows: all Spark jobs are run on a 16-node m2.4xlarge cluster with LZF compression enabled and the less compressible Workload 1.

The results for this experiment are unexpected. The baseline of Spark running on ext3 is much faster than ext4 in both phases, while Spark with shuffle file consolidation enabled runs much faster on ext4 than on ext3. Our optimization actually *worsened* the performance of Spark on ext3 by a significant factor. Fortunately, the shuffle file consolidation on ext4 still has the lowest job completion time out of all other options, including Spark without consolidation on ext3. Although Spark performance on ext4 with consolidation does not appear to lead the other options by significant margins on this particular workload, we believe the performance gap will be amplified as we scale up the size of the input data, and thus the number of Reduce tasks needed.

Why is shuffle file consolidation much more effective in ext4 than in ext3? We offer a preliminary conjecture on the differences between the two file systems' allocation schemes. Ext3 is an extended version of the traditional UNIX file system with journaling support. Due to backward compatibility reasons, ext3 is not radically different from its predecessor, and therefore continues to maintain files in terms of inodes. Under this scheme, the number of I/Os increase, although logarithmically, with the file size. Because of this, large files are especially susceptible to fragmentation.

The main advancement from ext3 to ext4 is the introduction of extents. Extents, a range of continguous physical blocks, replace the block mapping scheme used in ext3 and its predecessors. In particular, a single ext4 inode can store up to 4 extents, each of which can map up to 128 MiB of contiguous space per disk block. This feature is especially favorable to large files, which no longer require many disk seeks to read. Backward compatibility is also a concern in ext4's design, and therefore ext4 continues to use the inode interface shared with the file systems that preceded it.

In our case, it is easy to see why shuffle file consolidation offers a more significant performance improvement on ext4 than on ext3. Rather than creating many small, scattered shuffle files, we create much fewer, larger files. On ext3, these larger files may not fit into the first 12 direct address blocks on the inode, and therefore may have to rely on single, or even double, indirect address blocks, inflating the number of I/O's needed per file read. Further, there is no guarantee that the physical blocks representing the file are physically contiguous, though ext3 makes an effort to achieve physical locality.

In contrast, extents in ext4 provide much stronger physical locality benefits for large files. In particular, each large shuffle file can be read almost sequentially in its entirety, provided that it fits into an extent.

On the other hand, our data also suggests that the baseline Spark (without optimization) is significantly slower on ext4 than ext3. This appears to demonstrate a limitation of ext4 in handling a large number of small files.

6. FUTURE WORK

Our work on analyzing the current bottlenecks and the impact of shuffle file consolidation is still very much a work in progress. We intend to examine the kernel behavior more closely to discover exactly why shuffle file consolidation causes a slowdown in the ext3 file system. We additionally need to assess the new bottlenecks with the removal of the operating system file overheads. In the workloads examined, we believe the new bottleneck to be CPU, as it remains between 400% and 800%, while there are only 4 physical cores per node. This is likely due to the relatively high overhead of parsing and disassembling JSON inputs, so a workload that does less computational work may reveal bottlenecks in the other resources (network or disk).

With 16 nodes, we did not see TCP incast or network buffers as a significant issue. We suspect, however, that these issues would arise with much larger clusters. Although a surprising number of Spark clusters run between 4 and 16 nodes, our work does not reveal the nature of the higher end clusters of 100s or 1000s of nodes.

The shuffle operation performed by MapReduce has been stable for about four years now. While we compared our results in an end-to-end manner with MapReduce, a more direct comparison with MapReduce's shuffle algorithm could be beneficial, as it produces even fewer actual files at the cost of requiring a sort (but this sort can be used for future operations such as streaming joins).

Similarly, we could investigate the option of pushing shuffle files prior to the completion of the Map phase, as MapReduce does. This has the potential to halve the total job time in an ideal situation where the Reduce phase takes no CPU work and the map and shuffle phases can be completely overlapped. Examining the actual performance implications in real situations could be very useful.

A final optimization to consider is performing a completely in-memory shuffle when the size of the shuffle data is less than the cluster's collective memory. Preliminary tests that used ramfs showed that the file number overhead in disk-based file systems is not present in ramfs (i.e, Spark with shuffle file consolidation ran at the same rate as without). Additionally, disk throughput and seek latency would become a non-issue. The only challenges with this approach are degrading gracefully when shuffle data exceeds memory and correctly allocating memory between the RDD cache, mappers/reducers, and the shuffle data. Failure to do so correctly could lead to out-of-memory errors.

7. CONCLUSION

By identifying the shuffle phase bottlenecks specific to Spark, we have explored several alternatives to mitigate the operating system overheads associated with these bottlenecks. The most fruitful of which is shuffle file consolidation, a simple solution that led to a 2x improvement in overall job completion time. In addition to demonstrating that the performance of Spark with this optimization is comparable with that of Hadoop, our data also suggests that the performance of Spark scales linearly with the number of nodes.

We have submitted a patch to Spark for shuffle file consolidation. However, due to the performance regression on ext3, the feature is not enabled by default. The fact that Amazon EC2, a very popular environment for distributed computation, provides ext3 as the default file system presents a non-trivial hurdle for Spark users to exploit the performance improvement of this optimization. A short term solution is to change the Spark EC2 scripts to automatically mount ext4 on each machine on cluster start-up, an effort already in progress. A longer term solution is to further investigate the root cause of performance regression and instrument a change in the optimization code itself to account for file systems that do not rely on extents.

8. REFERENCES

[1] Yanpei Chen, Rean Griffith, Junda Liu, Randy H Katz, and Anthony D Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 73–82. ACM, 2009.

[2] Apache hadoop. http://apache.hadoop.org.

[3] Alexander Rasmussen, George Porter, Michael Conley, Harsha V Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. Tritonsort: A balanced large-scale sorting system. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 3–3. USENIX Association, 2011.

[4] Scalding. http://github.com/twitter/scalding.

[5] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[6] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.